Variable Precision Floating-Point RISC-V Coprocessor Evaluation using Lightweight Software and Compiler Support

Tiago T. Jost*
CEA, LETI, Univ. Grenoble Alpes
Grenoble, France
tiago.trevisanjost@cea.fr

Andrea Bocco*
CEA, LETI, Univ. Grenoble Alpes
Grenoble, France
andrea.bocco@cea.fr

Yves Durand
CEA, LETI, Univ. Grenoble Alpes
Grenoble, France
yves.durand@cea.fr

Christian Fabre
CEA, LETI, Univ. Grenoble Alpes
Grenoble, France
christian.fabre@cea.fr

Florent de Dinechin
INSA
Lyon, France
florent.de-dinechin@insa-lyon.fr

Albert Cohen
Google
Paris, France
4c0h3n@gmail.com

ABSTRACT

The popularity and community-driven development model of RISC-V have opened many areas of investigation to researchers and engineers. To overcome some of the IEEE 754 standard's limitations, one currently emerging avenue for computer architecture and systems research is the area of alternative floating-point computation. The UNUM format, for instance, offers variable precision and much flexibility useful to scientific computing or computational geometry. Programmers usually rely on arbitrary precision libraries such as MPFR (itself depending on GMP). However, there is currently no specialized RISC-V support for these libraries, and little support for variable precision arithmetic across the tool chain in general. We propose a framework to explore the potential of variable precision arithmetic in scientific computing applications on RISC-V processors. This work comprises: (i) a floating-point RISC-V coprocessor which improve accuracy using the UNUM format; (ii) an ISA extension of the RISC-V ISA for the unit, (iii) a programming model for this extension, and (iv) RISC-V optimized routines for the GMP library. Comparing our solution with MPFR on linear systems solvers, we are able to achieve speedups of up to 18× while keeping computational errors within the same order of magnitude. For 512 bits of precision, speedup between 9x and 16x are observed.

CCS CONCEPTS

• **Computer systems organization** → Reduced instruction set computing; Embedded hardware; Embedded software.

KEYWORDS

Variable precision, Floating-point, UNUM, Scientific computing, Coprocessor, Multiple precision, MPFR, GMP, GCC, LLVM

1 INTRODUCTION

Computer systems conform to the IEEE 754 standard [17] as the default format for floating-point (FP) arithmetic. Proposed in the 1980s and revisited in 2008, the standard has been successful in most application domains. However, a widening range of scientific computing applications do not comply with this format, due to their need of larger, smaller, and/or adaptive precision during the course of their computation, up to hundreds of digits [2, 3].

*Both authors contributed equally to this research.

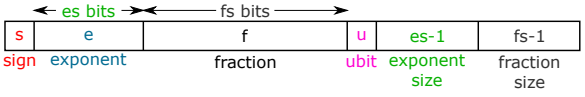

Figure 1: The Universal NUMber (UNUM) Format

These applications suffer from cancellation and rounding errors; increasing precision may significantly improve stability, convergence, or compensate for ill-conditioning. To address these issues users often rely on multi-precision libraries, such as MPFR [6] or GMP [7], to achieve satisfactory accuracy. In addition, one alternative emerging research venue lies into rethinking FP arithmetic through new representations. The UNUM format [8] is an alternative Variable Precision (VP) FP representation which offers variable length exponent and mantissa field and that may be suitable for scientific computing and other domains.

In this context, the open environment of the RISC-V architecture is an ideal instrument to evaluate the effectiveness of these upcoming formats, and to exploit the most promising ones. Therefore, we propose a framework to explore the potential of VP arithmetic in scientific computing applications on RISC-V processors. This work extends the one done by Bocco et al. [4], which presented a multi-precision UNUM RISC-V coprocessor for scientific computing. The main contributions of this paper are the following:

(1) A new `vpfloat` primitive type at C level so that programmers have access to the coprocessor capabilities.
(2) An Application Binary Interface for the proposed ISA [4].
(3) An optimization of GMP library for RISC-V, used in MPFR.
(4) A benchmark comparison between the proposed hardware solution and MPFR for three linear system solvers.

The remaining of this work is organized as follows: Section 2 gives an overview of the UNUM format. Section 3 outlines the RISC-V coprocessor and the ISA extension for the unit, and Section 4 discusses some of the software aspects of VP. Section 5 presents the experimental results when comparing coprocessor acceleration with MPFR/GMP. Section 6 concludes this work.

2 UNUM FORMAT

Variable Precision (VP) computing is meant to compensate the limitations of the IEEE 754 Floating Point (FP) format [9] due to the accumulation of cancellation and rounding errors during algorithm

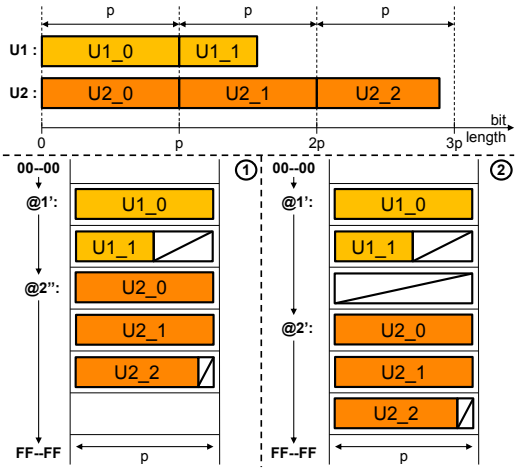

Figure 2: Variable length numbers in memory

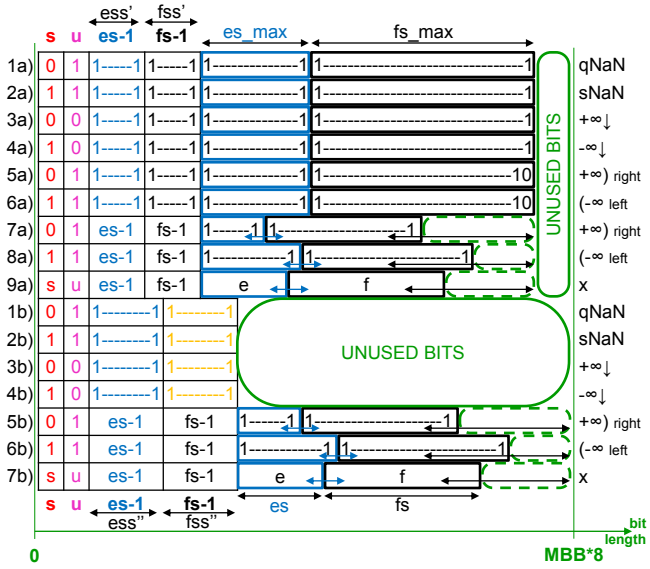

Figure 3: BMF: The Bounded Memory Format

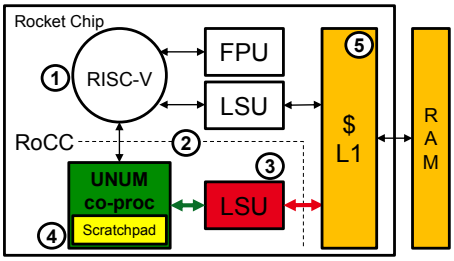

Figure 4: UNUM unit's architecture overview

execution. Nowadays VP is mainly implemented by means of software libraries (e.g. [7] and [6]). Some hardware alternatives were proposed [11, 14] but none of them address how to store VP FP numbers in main memory.

As far as we know, the UNUM FP format [8] (Figure 1, also as UNUM type I) is the only available variable-length FP format in the state of the art which has a dynamic representation for exponent and mantissa. This format self encodes the length of the exponent and fraction fields and it supports interval arithmetic (IA). IA can improve numeric calculations by doing a precise estimation of the computational error. We reorganized the UNUM fields (Figure 1) in order to have all the fixed length fields in the less significant part of the number (7b in Figure 3). In that way during the load operation it is possible to compute the actual length of the number.

We propose two addressing modes in memory where every VP FP data is seen as a chain of p-bit *chunks*. The granularity of p depends on the memory subsystem specifications (in RISC-V, p=8 bits). The first addressing mode ①, Figure 2, supports *compact arrays* in main memory with sequential memory accesses. For this, our proposed ISA [4] offers load and store instructions which return the address of the first chunk after the accessed element (e.g. the access of @1' returns @2'). Since it is impossible to compute, at compile time, the address of a random position of a compacted array element, elements overwriting is not allowed. This addressing mode is impractical in iterative applications, but it can be used to store arrays in main memory for a long period of time without losing precision on array elements.

The second addressing mode ② *aligns* the array elements on fixed-size slots. The *slot* size is a multiple of p-bits large enough to host the maximum bit length of array elements. Thus the array slot size ($3p$) and the array elements addresses (*@1'*, *@2'*) do not depend on the data and can be computed at compile time. In both ① and ② there might be some unused bits in memory (empty boxes ⧄).

In our system the developer can program (at the granularity of p bits), according to the application needs, the maximum VP FP numbers slot size for the second addressing mode ② by setting the *Maximum Byte Budget* (MBB) status register. Store operations re-round the UNUM format with MBB Bytes boundaries. Figure 3 depicts the proposed Bounded Memory Format (BMF) which remaps all the existing UNUM encodings [8] according to the MBB value. If the MBB value is larger or equal than the maximum UNUM bit-length (cases 1a-9a), according to its (ess', fss') Unum Environment (UE, [8]), the data is stored as it is (like in ②). If the MBB value is smaller than the maximum UNUM bit-length (case 1b-7b) specified in its UE (ess", fss"), BMF is applied: special values 1a-4a are mapped as 1b-4b, and all the other values are rounded (depending the rounding policy) in 7b. Infinities generated during the BMF rounding are mapped in 5b-6b. The BMF concept can be extended to all the self-descriptive variable-length VP FP formats.

3 UNUM COPROCESSOR AND ISA

We propose, from [4], an arithmetic Variable Precision (VP) Floating Point (FP) unit. It supports three format modus of operations through a dedicated Load and Store Unit (LSU): (i) For standard calculation and IO, we support standard IEEE 754 formats; (ii) For internal operations, we rely on 32 internal registers with programmable mantissa size (up to 8 64-bits chunks); (iii) For the intermediate

storage in memory, we adopt the UNUM layout specified in Section 2. Since memory space is a scarce resource, our LSU supports non-aligned memory accesses aligning data in memory on bytes.

Figure 4 depicts the computing system based on the RISC-V Rocketchip architecture [1]. The main core ①, generated with the Rocketchip generator, is connected to its native system: its 64 bits Floating Point Unit (FPU), its memory hierarchy and its peripherals. The system can host up to four coprocessors connected through a dedicated interface ②. Our system configuration has only one coprocessor ④ which hosts our UNUM accelerator.

According to the adopted UNUM (or ubound, UNUM intervals [8]) format, the data size information in memory is encoded in the data itself. Thus, coprocessor load instructions do not read constant-sized values from memory, like regular FP instructions. They read as many bytes as specified in the data descriptor fields in memory. Store instructions use the status register information, explained in Section 2, to write MBB bytes in memory. This property makes precision independent from the proposed Instruction Set Architecture (ISA). Additionally, precision in arithmetic operations also has the same properties, i.e., it is not controlled by the instructions, but through a status register. In other words, the coprocessor uses runtime information to determine the precision used during memory and arithmetic operations.

The coprocessor design is fully parametric. The one presented here is pipelined with 64-bits internal parallelism. Each pipeline stage implements a stop and wait protocol to process gbound (intervals, [8]) mantissas divided in chunks of 64 bits each.

The coprocessor scratchpad ④, the gbound Register File (gRF), hosts 32 intervals on the gbound format, which is different from the main memory ⑤ one (UNUMs, ubounds). Each interval has two endpoints divided in header and mantissa. Mantissas are always normalized (i.e. hidden bit implicitly set) and they are divided in 8 chunks (i.e. up to 512 bits). The header is made of sign, exponent, flags (NaN, ∞, zero, ...) and length (L) fields. L hosts the number of chunks used to encode the mantissa. The conversion between the gbound and UNUM/ubound formats is handled by a dedicated LSU, ③ during memory operations.

The coprocessor has four configuration Status Registers (SR): MBB (Section 2), DUE, SUE and WGP. The Default and the Secondary UNUM Environment (DUE and SUE) SRs host the UEs to be used during load/store UNUM/ubound operations. They are used to encode the length of data in memory (like MBB) and to speedup memory operations among different UEs (e.g. UE conversion). The Working G-layer Precision (WGP, [4]) SR hosts the maximum number of mantissa chunks that a coprocessor operator can output: the result output length (L) is constrained at the WGP SR value. For more details about the hardware system and the very efficient load and store interface hiding between the hardware coprocessor and main memory, please refer [4].

3.1 The coprocessor ISA

The coprocessor Instruction Set Architecture (ISA), Table 1, is an extension of the RISC-V one. The instruction field names are listed in ⓪. The supported operations (⑫-⑭) are comparisons, addition, subtraction, multiplication, interval midpoint (GGUESS) and interval radius (GRADIUS). Other operations (e.g. division) are implemented in software. The ISA has three main features: (i) It supports, by setting internal status registers, simultaneously different UEs and internal operation precisions (①-④); (ii) It supports internal registers copies and on-the-fly conversion among IEEE and gbound formats (⑤-⑪); (iii) It supports a dedicated Load and Store Unit (LSU) which handles misaligned memory accesses (⑮-⑱) for all the supported addressing modes Section 2. To use the SUE SR in load/store operations, an additional '_S' must be added after the ⑮-⑱ operation's names (e.g. `ldub_s`, `stul_s_next`, ...). For more details about the ISA functionalities please refer [4].

	31 25	24 20	19 15	14	13	12	11 7	6 0
⓪	func7	rs2	rs1	xd	xs1	xs2	rd	opcode
	7	5	5	1	1	1	5	7
①	susr	unused	Xs1	0	1	0	unused	CUST
②	lusr	unused	unused	1	0	0	Xd	CUST
③	smbb/swgp/sdue/ssue	unused	Xs1	0	1	0	unused	CUST
④	lmbb/lwgp/ldue/lsue	unused	unused	1	0	0	Xd	CUST
⑤	mov_g2g	unused	gRs1	0	0	0	gRd	CUST
⑥	movll/movlr	unused	gRs1	0	0	0	gRd	CUST
⑦	movrl/movrr	unused	gRs1	0	0	0	gRd	CUST
⑧	mov_x2g	#imm5	Xs1	0	1	0	gRd	CUST
⑨	mov_g2x	#imm5	gRs2	1	0	0	Xd	CUST
⑩	mov_d2g/mov_f2g	#imm5	Xs1	0	1	0	gRd	CUST
⑪	mov_g2d/mov_g2f	#imm5	gRs2	1	0	0	Xd	CUST
⑫	fcvt.x.g/fcvt.g.x	unused	Xs1	0	1	0	gRd	CUST
⑬	fcvt.f.g/fcvt.g.f	unused	Xs1	0	1	0	gRd	CUST
⑭	fcvt.d.g/fcvt.g.d	unused	Xs1	0	1	0	gRd	CUST
⑮	gcmp	gRs2	gRs1	1	0	0	Xd	CUST
⑯	gadd/gsub/gmul	gRs2	gRs1	0	0	0	gRd	CUST
⑰	gguess/gradius	unused	gRs1	0	0	0	gRd	CUST
⑱	lgu/ldub	unused	Xs1	0	1	0	gRd	CUST
⑲	stul/stub	gRs2	Xs1	0	1	0	unused	CUST
⑳	lgu_next/ldub_next	gRs2	Xs1	1	1	0	Xd	CUST
㉑	stul_next/stub_next	gRs2	Xs1	1	1	0	Xd	CUST

Table 1: Coprocessor's Instruction Set Architecture

4 VARIABLE PRECISION SOFTWARE AND COMPILER SUPPORT

Software must contribute to ensure that newly architectures and designs are accessible at code level, granting developers access to the most advanced hardware resources. For example, CUDA [10] and OpenCL [13] are widely used for programming Graphics Processing Units (GPUs), and parallelism in multi-core and many-core processors can be explored through OpenMP [5] and MPI [15] libraries. Similarly, we also provide a simple and intuitive way to use the VP capabilities of our coprocessor. The remainder of this section focuses on the software support for VP and the unit, covering aspects from the Application Binary Interface to the compilation. Additionally, we provide a code example to illustrate the use of the data type and the ISA.

4.1 Coprocessor ISA ABI

The Application Binary Interface (ABI) specification for the ISA extension is similar to the standard RISC-V FP ABI. Table 2 lists the coprocessor registers and respective roles in the defined calling convention. Register naming uses the same convention as the FP registers in RISC-V. However, in lieu of using the f as prefix, letter g is used to denote coprocessor registers. The calling convention for argument and return values also respects RISC-V FP ABI where `g10`

Register	ABI Name	Description	Saver
g0-7	gt0-7	Temporaries	Caller
g8-9	gs0-1	Saved Registers	Callee
g10-11	ga0-1	Arguments/return values	Caller
g12-17	ga2-7	Arguments	Caller
g18-27	gs2-11	Saved Registers	Callee
g28-31	gt8-11	Temporaries	Caller

Table 2: ABI Convention for the VP registers

```
1 vpfloat factorial(unsigned k) {
2     unsigned old_wgp = set_precision(320);
3     vpfloat fact = 1.0;
4     for (int i = 1; i < k+1; ++i) {
5         fact *= i;
6     }
7     set_precision(old_wgp);
8     return fact;
9 }
```

Example 1: Usage of the variable precision unit in C code

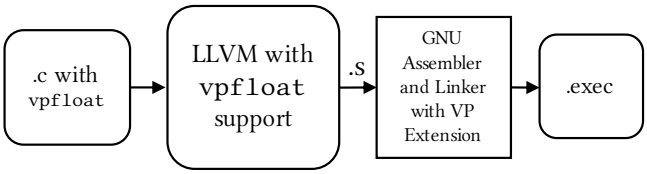

Figure 5: Compiling VP applications

and g11 are used, as well as the registers preserved across function calls.

4.2 New data type support

Like the `float` and `double` data type semantics, we propose a new `vpfloat` primitive type to allow the use of the VP coprocessor. In this first version of the compiler, these variables are encoded on 36 bytes fixed-slots in memory in such a way variables with 256 bits of precision can be stored in memory as required by the coprocessor. Note that the UNUM format requires extra bits (sign, ubit, and metadata fields), which can add up to 4 bytes on the representation. The user can manually modify the coprocessor internal precision (WGP) by using the assembly instruction `susr` and `swgp`.

Example 1 shows how to calculate the factorial of numbers using VP in C. This algorithm tends to generate numbers with high orders of magnitude, so it is fitted as a practical example of how VP can be used in code. The algorithm starts by saving the previously used WGP value and setting a new value of 320 bits of mantissa precision (at line 2). The `for-loop` calculates the factorial and accumulates the value in variable `fact` (lines 4 to 6). Before leaving the function, the previous value of WGP is restored.

A new type was added to the LLVM [12] Frontend and Type System so that `vpfloat` is recognized from frontend to backend. Since the Type System requires a fixed size represention in the middle-end, `vpfloat` size was set to 36 bytes, which is enough to hold the maximum UNUM format available on the coprocessor.

```
1         ...
2         call    set_precision
3         addi    a1, s1, 1
4         addi    a2, zero, 2
5         bltu    a1, a2, .LBB0_3
6 # %bb.1:                                # %for.body.preheader
7         lla     a1, (.Constant_1)
8         lgu     gt2, (a1)
9         lla     a1, (.Constant_1)
10        lgu     gt0, (a1)
11        lla     a1, (.Constant_1)
12        lgu     gt1, (a1)
13 .LBB0_2:                               # %for.body
14        gmul    gt2, gt2, gt0
15        gguess  gt2, gt2
16        gadd    gt0, gt0, gt1
17        gguess  gt0, gt0
18        addi    s1, s1, -1
19        bnez    s1, .LBB0_2
20        j       .LBB0_4
21 .LBB0_3:
22        lla     a1, (.Constant_1)
23        lgu     gt2, (a1)
24 .LBB0_4:                               # %for.cond.cleanup
25        lui     a1, 0
26        addi    a1, a1, -256
27        add     a1, s0, a1
28        stu     gt2, (a1)
29        call    set_precision
30        lui     a0, 0
31        addi    a0, a0, -256
32        add     a0, s0, a0
33        lgu     ga0, (a0)
34        ...
```

Example 2: Snippet of the assembly code for Example 1

The RISC-V LLVM backend was extended to support the new ISA extension illustrated in Table 1 and the new data type. Additionally, the RISC-V GNU Assembler and Linker were expanded to generate executable code for the coprocessor ISA extension. Figure 5 shows the compilation flow for VP applications.

Example 2 shows a snippet of the assembly code generated by LLVM for Example 1. Coprocessor instructions are found at lines 8, 10, 12, 14-17, 23, 28 and 33. The code starts by calling the external function `set_precision` (line 2) which sets the WGP to 320 bits and returns the previous WGP value. From line 7 to 12, coprocessor registers `gt0-gt2` are initialized to 1, and the `for-loop` (lines 13-20) calculates the factorial values through `gmul` and `gadd` instructions. As explained in Section 3, the coprocessor operates with interval numbers as it adopts the UNUM format. Since `vpfloat` represents scalars, `gguess` instructions (lines 15 and 17) are necessary after every arithmetic operation so that the middlepoint of the interval is obtained. In the last basic block, WGP is reset to its previous value (line 27) and the calculated value is passed to the return register `ga0` (line 31). Moreover, we notice how the result is stored in memory (lines 25-28) and loaded back to return register `ga0` (lines 30-33), since register `gt2` is not preserved during function calls.

5 EXPERIMENTAL RESULTS

In this section, we present the methodology used for our experiments and the results obtained when comparing our solution to a software-based approach. We also cover an optimized GMP for RISC-V used to accelerate baseline applications.

Routines	Nº Instructions (Compiler)	Nº Instructions (Specialized Asm)
umul_ppm	12	3
mul_1	25	11

Table 3: GMP specialized routines

5.1 Experimental setup

We have selected three matrix-based linear solvers as cases of study for Variable Precision (VP): 1/ *Gauss elimination* (GE), 2/ the *Conjugate gradient* (CG) and 3/ *Jacobi* (JA). These algorithms execute long chains of multiply-addition operations that tend to accumulate errors, so they are suitable for experimentation. Three different matrices were selected for performance and error evaluation: a Hilbert 15x15 matrix (*Hilbert*), a random-generated 24x24 matrix (*random*) and a 40x40 dominant diagonal matrix (*diag dom*). Applications *GE* and *CG* were compiled and run for all three configurations, while JA used only diagonal-dominant matrix due to algorithm constraints.

Our VP coprocessor [4] is attached to a 64-bit RISC-V main processor hosted in the Rocket Chip system. The choice for a 64-bit processor is due to being a more realistic scenario for High-Performance Computing (HPC) environments with large data sets. The system was built on top of a Xilinx Virtex-7 FPGA and coprocessor applications are executed in a baremetal environment. As baseline for our experiments, we have implemented the same benchmarks using the MPFR library, as no hardware solution is available. Section 5.2 presents further information on specialized code to accelerate applications written in MPFR. These applications were executed in a cycle-accurate emulator from the Rocketchip project [1], since our FPGA setup had no OS support. We have compiled VP and MPFR applications with LLVM, like described in 5. Moreover, MPFR and GMP libraries were compiled using the GNU Compiler (GCC), as it has been a more stable compiler tool chain for RISC-V.

5.2 Optimized GMP Library for RISC-V

Programmers usually rely on the GMP [7] and MPFR [6] libraries to explore VP in applications. Although these libraries can be easily compiled for RISC-V systems, it is possible to accelerate them by implementing assembly routines with specialized code.

In order to have fair hardware and software comparison between our coprocessor and MPFR we have chosen to implement some GMP routines in RISC-V assembly, so that GMP/MPFR applications run faster. Since MPFR uses GMP routines underneath, the same routines implemented for GMP are also used for MPFR applications. The purpose of extending GMP with RISC-V specialization is twofold: (i) provide a fairer comparison between our coprocessor-based approach and software-based solutions and (ii) encourage the community to contribute with RISC-V specialized code for GMP by making these routines available to the community.

Table 3 shows the list of routines implemented in assembly, with a comparison on the number of instructions for the main for-loop of each routine. Particularly, `umul_ppm` and `mul_1` were selected because they are highly used for multiplication, division, square root and power functions inside GMP. With a simple, yet efficient, implementation we are able to reduce the number of instructions of the routine's main loop in comparison to the code generated by

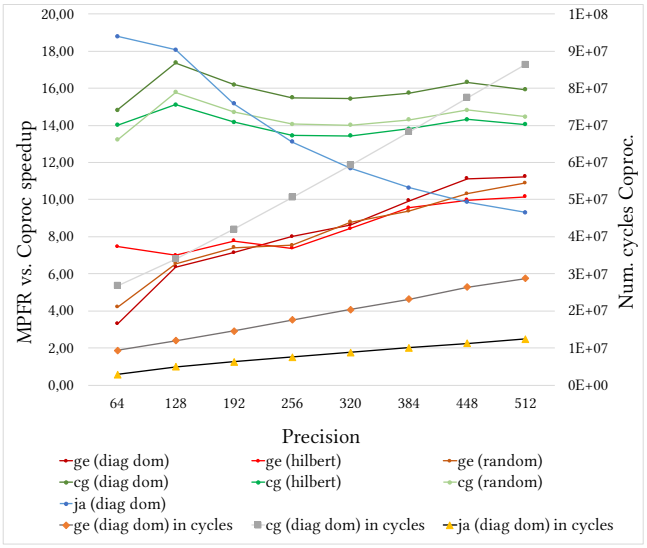

Figure 6: MPFR versus VP Coprocessor

RISC-V GCC with O3 optimization. The GCC code is not able to infer instruction `mulhu` [16] that performs multiplication and returns the higher bits of the product. A simple implementation using `mulhu` improves performance of the routines considerably (as shown in Table 3). We have also implemented `add_n` and `sub_n` routines to handle propagation of overflow and underflow flags, however, they have shown no performance improvements in comparison to the GCC O3 baseline. Moreover, we also plan on adding support to other routines so performance on GMP improves even more.

5.3 Performance

Colored lines in Figure 6 depicts the speedup achieved by our coprocessor in comparison to MPFR (left y axis) varying the mantissa precision (WGP for the coprocessor). Grey lines shows the number of clock cycles executed in the coprocessor (right y axis). We notice how the speedup varies according to the application and precision used. The speedups are in function of the algorithm types, while the matrices sizes and the types have little influence on the speedup.

As mentioned in Section 5.4, the coprocessor supports mantissas of up to 256 bits in memory (only in the coprocessor scratchpad the computation can rise up to 512 bits). For that reason, after 256 bits of precision, coprocessor performance results cannot be fairly compared with the MPFR ones. The coprocessor latency increases linearly with the mantissa precision augmentation. This linearity is broken at 256 bits of mantissa (the slop of the grey curves changes) since we do not load and store values with higher precision.

Our solution shows a clear advantage over the baseline. This advantage comes from the native hardware support for handling large FP numbers. While MPFR applications access memory constantly to read part of the number for calculation, we make use of a register file that can hold up to 32 values. Additionally, our coprocessor, thanks to its pipelining capabilities, is able to execute multiple operations in parallel.

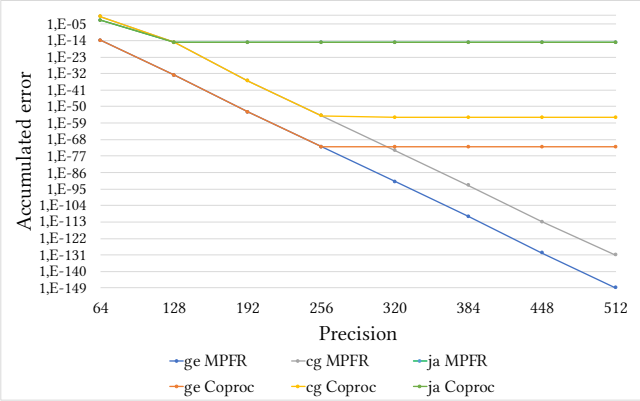

Figure 7: Error Analysis for the three applications

5.4 Error Analysis

The most important criterion for changing the precision used is the accumulation of error during computation. Execution time is significantly impacted when precision is increased: the output error must be correlated to the increasing of the application precision.

Figure 7 shows the results of the error analysis obtained for the three applications. We notice that *coprocessor* and *MPFR* accumulated errors have the same order of magnitude between 64 and 256 bits of precision, proving that we are able to improve performance on applications without impacting the accumulated error. They diverge above 256 bits because, as previously explained, the coprocessor load/store unit is limited to 256 bits, while MPFR applications can go beyond the 256-bit cap. After this consideration we plan to extend the coprocessor load and store unit to support up to 512 fractional bits in main memory.

6 CONCLUSION

We proposed a framework for the exploration of variable precision arithmetic in scientific computing applications on RISC-V processors. The proposed solution comprises a VP FP coprocessor, ISA extension, programming model and RISC-V port of the GMP library. Results have shown, for similar computational error, speedups between 3.5× and 18× in comparison to MPFR for three different applications for solving linear systems. For 512 bits of precision, we have observed speedup between 9× and 16×.

We envision as future work to extend the use of VP to other domains in order to better understand the requirements for VP in general. We will make the GMP library improvements available to the community by the time of the workshop and will continue exploring GMP optimizations for RISC-V processors.

ACKNOWLEDGMENTS

This work was partially funded by the French *Agence nationale de la recherche (ANR)* for project IMPRENUM (Improving Predictability of Numerical Computations) under grant n° ANR-18-CE46-0011.

REFERENCES

[1] 2016. *The Rocket Chip Generator.* Technical Report UCB/EECS-2016-17. EECS Department, University of California, Berkeley. http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-17.html

[2] David H Bailey. 2005. High-precision floating-point arithmetic in scientific computation. *Computing in science & engineering* 7, 3 (2005), 54–61.

[3] David H Bailey and Jonathan M Borwein. 2015. High-precision arithmetic in mathematical physics. *Mathematics* (2015), 337–367.

[4] A. Bocco, Y. Durand, and F. de Dinechin. 2019. SMURF: Scalar Multiple-precision UNUM RISC-V Floating-point Accelerator for Scientific Computing. In *CoNGA*. https://doi.org/10.1145/3316279.3316280

[5] Rohit Chandra, Leo Dagum, David Kohr, Ramesh Menon, Dror Maydan, and Jeff McDonald. 2001. *Parallel programming in OpenMP.* Morgan kaufmann.

[6] Laurent Fousse, Guillaume Hanrot, Vincent Lefèvre, Patrick Pélissier, and Paul Zimmermann. 2007. MPFR: A Multiple-precision Binary Floating-point Library with Correct Rounding. *ACM Trans. Math. Softw.* 33, 2, Article 13 (June 2007). https://doi.org/10.1145/1236463.1236468

[7] Torbjörn Granlund and the GMP development team. 2012. *GNU MP: The GNU Multiple Precision Arithmetic Library.* https://gmplib.org/ Version 5.0.5.

[8] John L Gustafson. 2015. *The End of Error: Unum Computing.* Chapman and Hall/CRC.

[9] IEEE754-2008 2008. IEEE Standard for Floating-Point Arithmetic. IEEE 754-2008, also ISO/IEC/IEEE 60559:2011. https://doi.org/10.1109/IEEESTD.2008.4610935

[10] David Kirk et al. [n. d.]. NVIDIA CUDA software and GPU parallel computing architecture.

[11] Ulrich Kulisch. 2018. *Computer arithmetic and validity: Theory, implementation, and applications.* Berlin, Boston: De Gruyter.

[12] C. Lattner and V. Adve. 2004. LLVM: a compilation framework for lifelong program analysis transformation. In *Intl. Symp. on Code Generation and Optimization (CGO)*. 75–86.

[13] Aaftab Munshi. 2009. The opencl specification. In *2009 IEEE Hot Chips 21 Symposium (HCS)*. IEEE, 1–314.

[14] M. J. Schulte and E. E. Swartzlander. 2000. A family of variable-precision interval arithmetic processors. *IEEE Trans. Comput.* 49, 5 (May 2000), 387–397. https://doi.org/10.1109/12.859535

[15] Marc Snir, William Gropp, Steve Otto, Steven Huss-Lederman, Jack Dongarra, and David Walker. 1998. *MPI–the Complete Reference: The MPI core.* Vol. 1. MIT press.

[16] Andrew Waterman and Krste Asanović. 2017. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA.* RISC-V Foundation.

[17] Dan Zuras, Mike Cowlishaw, Alex Aiken, Matthew Applegate, David Bailey, Steve Bass, Dileep Bhandarkar, Mahesh Bhat, David Bindel, Sylvie Boldo, et al. 2008. IEEE standard for floating-point arithmetic. *IEEE Std 754-2008* (2008), 1–70.